SICStus maskinkodskompilering

Kent Boortz
Swedish Institute of Computer Science
Box 1263, S-164 28 Kista, Sweden

15 augusti 1991

T91:13

Sammanfattning

SICStus Prolog bygger på en WAM-emulator, *Warrens Abstract Machine*, skriven i C för portabilitet. Denna rapport beskriver SICStus maskinkods-utökning som gör att predikat kan kompileras direkt till maskinkod på maskiner baserade på 68K- och Sparc-processorer.

1. Inledning

Följande text beskriver strukturen på de delar i SICStus Prolog 2.0 som rör utökningen för maskinkodskompilering. SICStus kompilator och emulator finns beskriven i andra rapporter [Car90:1] [Car90:2] [Car91].

◊ Detta är ingen fristående rapport. Läsaren förutsätts ha tillgång till källkoden till SICStus och inblick i hur systemet fungerar.

2. Kompilatorn

SICStus kompilator arbetar i flera steg. I den följande beskrivningen avser *kompilatorn* den del som kompilerar från Prolog till WAM. *Kodgenerering* avser kompileringen från WAM till ”byte”-kod, maskinkod, ”quick load”-fil eller symbolisk WAM-fil.

3. Prolog till WAM plwam.p4

Ändringarna i den del som kompilerar från Prolog till WAM är relativt små. Maskinkods-utökningen använder inte ”shallow backtracking” [Car90:2] så WAM-instruktioner speciellt för detta faller bort. Registerallokeringen skiljer därför något. Speciella instruktioner för att stödja ”Full Write Mode Propagation” har införts [Bor91].

◊ Man kan inte skapa den symboliska WAM-koden med *plwam/2* och använda den som utgångspunkt vid felsökning av maskinkod. Sätt istället ”spy points” på predikaten *prolog:nc_compile_predicate/4* och *prolog:nc_compile_clause/5*.

4. Drivrutiner för kodgenerering *wamql.p4 wamnql.pl*

Snitten mellan kompilatorn och de olika kodgeneratorerna är likartade. Kompileringsordningen skiljer sig något mellan olika kodslag. Vid maskinkods-kompilering kompileras klausulerna i ett

predikat först och sedan skapas kod för indexering. Vid byte-kods-kompilering behövs ingen indexeringskod.

5. Kod för indexering *n68_pred.p4 nsp_pred.p4*

I emulatorn motsvaras koden för indexering av en struktur i minnet som avkodas vid anrop. I maskinkodsversionen skapas maskinkod för indexering.

Stegen är följande:

1. **Testa om ett ”event” av något slag har inträffat.**
   Det kan vara "heap overflow", "spy point" eller något annat som måste hanteras innan själva predikatanropet. I så fall anropas *event_emul,* en rutin som lämnar över till emulatorn. Emulatorn kommer då att hantera ”events” och sedan försöka emulera predikatet men upptäcker att den är i maskinkod och anropar maskinkod. Vi kommer då till samma punkt som vi startade vid men alla ”events” är avklarade.

   ```
   __event_emul:                    ! SPARC version
      sub  t0,d_entry,t0
      st   t0,[w+insn]              ! w->insn = P
   Levent:
      NATIVE_TO_EMUL
      XREGS_TO_WORKER
      RESTORE_C
      retl                          ! return to emulator
      mov  EXECUTE_EMUL,%o0         ! return value
   ```

   Observera att om anropet kommer från en maskinkodsklausul i SPARC-versionen så hoppas de två första instruktionerna av indexeringen över. Dessa instruktioner ligger som dubletter i de två föregående ”delay slots” och har redan körts (se rubriken ”Snittet mot emulatorn” och ”Kodgenerering av klausuler–Kontroll”).
2. **Register X0 derefereras.**
3. **Typen avgörs.**
4. **Rätt klausul(er) letas fram.**
   Genom jämförelse eller hashning (se nedan).
5. **Anrop av klausul.**
   Om det bara finns en klausul som passar sker ett hopp till denna, annars skapas en valpunkt och det sker ett hopp till den första klasulen i ordningen.

5.1. Hashning

Hashning sker på små heltal, atom-pekare, och funktor-pekare Små heltal och atom-pekare kan samsas i samma hash-tabell. I struktur-fallet är det inte strukturpekaren, vilken inte är unik, utan funktorn som hashning sker på.

Hashtabellen har storleken en jämn tvåpotens större än antalet element. Kollisioner hanteras med länkade kedjor. Varje element i tabellen har formen

```
long(Key)
long(Value)
long(Next)
long(Previous)
```

Offset *Previous* används bara vid uppbyggandet av tabellen. Offset *Next* pekar ut nästa element i kedjan. Alla element kommer att ligga i tabellen, dvs inget extra utrymme utöver tabellen behövs.

Noderna läggs vid genereringen in efter varandra i godtycklig ordning och utan att länkas ihop. Tomma noder har alla fält satta till *NIL*. Efter tabellen ligger länkinstruktionen *rehash* som talar om för Prolog-länkaren att ordna hashtabellen.

Hash-funktionen är enkel

Index = (TabSize -1) & (Key >> 2)

eller eftersom varje element är 4 ”long words”

Long_offset = (TabSize -1) << 2) & Key

Funktionen är vald så att små heltal som följer varandra inte får samma adress och därmed ingår i samma kedja.

5.2. Try-retry-trust

Try, det första alternativet, är ett direkt hopp till en klausul. *Retry-trust* är en kedja som läggs statiskt i minnet. Varje nod i kedjan ser ut som

```
long(Next)
long(Clause)
long(Fail)
short(Offset)   ! see emulator documentation
short(0)        ! alignment
```

Next pekar ut nästa element i kedjan. Om inga fler element finns så är den 0. *Clause* är adressen till klausulen. *Fail* är adressen till den fail-rutin som ska användas.

Om ariteten på predikatet är mindre än 13 så pekar detta fält ut en specialiserad fail-rutin som känner till ariteten. Det kommer också att vara olika rutiner för om det är den sista noden eller inte. Detta gör att fail-rutinen kan känna till om den ska ta bort valpunkten eller inte. Specialiserade fail-rutiner gör att backning i maskinkodsversionen blir mycket effektiv.

Det finns också specialiserade try-rutiner med avseende på aritet.

6. Kodgenerering av klausuler *n68_clause.p4 nsp_clause.p4*

Genereringen av kod för klausuler är i mycket en makro-expansion. En WAM-instruktion motsvarar en eller flera assembler-instruktioner. Vissa instruktioner expanderar till ett subrutin-anrop för att spara minne.

Strukturer i huvudet av en klausul hanteras med ”Full Write Mode Propagation” för att öka prestandan vid skapandet av större strukturer. Detta beskrivs i en separat rapport.

I 68K-fallet görs ett extra optimeringspass över den symboliska assemblerkoden.

6.1. Speciella X0-instruktioner

Det finns fyra startpunkter i koden för en klausul. Om klausulen anropas vid fail från emulatorn så emuleras koden. Före själva maskinkoden ligger en byte-kod, *RETRY_NATIVE*, så att emulatorn kan anropa maskinkod. Efter denna instruktion ligger startpunkten om koden anropas vid fail från maskinkod.

Det finns två startpunkter som anropas från indexeringskoden, en för ”Write mode” och en för ”Read mode” vilket motsvaras av om register X0 derefererades till en obunden variabel eller inte.

Om anropet kommer direkt från indexeringskod i maskinkod så är WAM register S, strukturpekaren, redan satt i aktuella fall. X0 är det derefererade taggade värdet och S det otaggade. Om anropet kommer från en fail-rutin så är X0 derefererad.

6.2. Inbyggda predikat och funktioner

Inbyggda predikat, ”Builtins”, ligger i subrutinbiblioteket.

```
nc_insn(builtin_1(Builtin,X), WC, WC) --> !,
   {nc_kernel(Builtin, Builtin1)},     % get routine number
   [move(x(X), t0),
    call(builtin(Builtin1))].
```

Kodgenerering för unära inbyggda predikat

Funktioner ligger också som subrutiner. Skillnaden är att returadressen kommer att peka ut data som ”garbage collectorn” behöver. Returvärdet läggs i register t0.

```
nc_insn(function_1(Func,Xi,Xj,Heap,Args), WC, WC) --> !,
   {nc_kernel(Func, Function)},        % get routine number
   [move(x(Xj), t0),
    call(builtin(Function)),
    long(Heap),
    short(Args),
    move(t0, x(Xi))].
```

Kodgenerering för unära inbyggda funktioner

6.3. Kontroll

I **SPARC-versionen** utnyttjas ”delay slots” efter hopp-instruktioner om möjligt Det händer ibland att detta kodutrymme inte kan fyllas med en instruktion förutom *nop* (no operation).

I fallet *execute(Predicate)* vet vi att en ny maskinkodsklausul med största sannolikhet kommer att köras och att den startar med ett test av ”events”.

```
nc_insn(execute(Label), _, []) --> !,
   [call(native_entry(Label)),         % P = Label
    sethi(hi(builtin(__heap_warn_soft)),t0)].
```

7. Assemblering *n68_asm.p4 nsp_asm.p4*

Assemblatorn skapar en struktur som sedan kan länkas in i minnet eller omvandlas till ql-format på fil. Att inte UNIX normala assemblator används beror på att Prolog-systemet har en egen symbolhantering som assemblatorn och länkaren inte känner till.

Assemblatorn gör ett pass baklänges och sedan ett litet pass med det som inte kunde göras första gången. Att den arbetar baklänges beror på att de flesta hoppen i koden är framåt och då löses snabbare och mer optimalt, speciellt i 68K-fallet.

- ◊ I *nsp_asm.p4* och *n68_asm.p4* finns information om vilket maskinregister som motsvarar vilket WAM-register och

temporärregister. Denna information måste stämma med motsvarande information i assembler-filerna *kernel_sp.s* och *kernel_68.s*.

8. Dynamisk länkning *objareas.c*

Den struktur som skapats av assemblatorn är en lista av Prolog-länkinformation. Att inte UNIX länkare används beror på att Prolog-systemet har en egen symbolhantering som länkaren inte känner till. Denna länkinformation används också för att skapa ql-filer.

◊ Koden som skapas kommer att vara exekverbar men ligga i UNIX-processens data-area istället för text-area (kod-area). Detta ger normalt inga problem men försvårar avlusning med adb och gdb eftersom det inte finns någon länkinformation och därmed inga symboler som visar vilken klausul eller vilket predikat som exekveras.

8.1. Prologs länkformat

Ett inbyggt predikat, *$make_native_code_object/3*, tar en prolog-lista av länkelement och skapar ett maskinkodsprogram i minnet.

Följande beskrivning av hur länkinformationen tolkas är lite ändrad mot den kod som finns i SICStus för att vara klarare att läsa. Variabeln *code* har för läsbarhetens skull ändrats till att vara unionen

```
union CODE {
   TAGGED *taggedp;
   CInfo cinfop;
   long *longp;
   short *shortp;
   char *charp, **charpp;
   unsigned long *ulongp, ulong;
} code;
```

Listan med länkinformation kan ha följande element

- Numeriskt, i SPARC-versionen är detta ett 32-bitars ord och i 68K-versionen ett 16-bitarsord.

```
case NUM:
#ifdef NC68K
      *code.shortp++ = (INSN)GetSmall(arg1);
#endif
#ifdef NCSPARC
      *code.longp++ = (long)GetInteger(arg1);
#endif
      break;
```

  Ett stort numeriskt värde lagras som en struktur på heapen och kan ha tag STR [Car91]. Detta är bara aktuellt i SPARC-versionen eftersom tal så små att de får plats i 16-bitarsord inte lagras på detta sätt.

```
case STR:
   {
      TAGGED func = TagToHeadfunctor(arg1);
#ifdef NCSPARC
      if(func & QMask)
         {
            DerefArg(arg1,1);
            *code.longp++ = GetInteger(arg1);
            break;
         }
#endif
```

- **Symbolisk**, en prolog-struktur med olika fält tolkas till ett värde eller tolkas som ett kommando att utföra en viss operation. Dessa beskrivs nedan.

8.2. Gemensamt för olika maskinkodsversioner

Dessa länk-instruktioner behövs både i SPARC- och 68K-versionen.

functor(Name,Arity) — skapar en funktor som ett 32-bitarsord.

```
*code.taggedp++ = SetArity(arg1,GetSmall(arg2));
```

tagged(Term) — sparar det taggade värdet

```
*code.taggedp++ = arg1;
```

builtin(Integer) — tar fram ett värde ur en tabell, *builtintab*, med index *Integer*.

```
*code.cinfop++ = builtintab[GetSmall(arg1)];
```

local(Address) — är ett lokalt offset inom en klausul. Eftersom assemblatorn assemblerar baklänges måste viss justering av värdet göras.

```
*code.charpp++ = (char *)object->emulcode + GetSmall(arg1)
                + GetSmall(clause_size);
```

absolute(Base,Offset) — tar fram den absoluta adressen av en klausul i predikatet som kompileras. Base är klausulnummer. Lägger till ett offset.

```
*(char **)code = clause_address(arg1) + GetSmall(arg2);
```

native_entry(Predicate) — tar fram adressen till koden som ligger i ett predikats definition, ”struct definition” (se nedan, Snittet mot emulatorn – Anrop av predikat).

```
DerefHeap(arg1, TagToArg(arg1,1));
*code.insnppp++ =
        (INSN **)&parse_definition(arg1, typein_mod)->entry;
```

rehash(table,tabsize) — tar en oordnad tabell och hashar in elementen på sina rätta platser.

```
rehash((char *)code - GetInteger(arg1) * sizeof(HashElement), i);
```

8.3. 68K-versionen

Det behövs olika länkdata för olika processorer. I 68K-versionen är instruktionerna av olika längd och ofta ligger det som måste ändras som ett separat 16- eller 32-bitsord.

long(Num) — tar fram 32-bitarstalet.

```
*code.longp++ = GetInteger(arg1);
```

builtin_short(Integer) — tar fram ett värde ur en tabell, *builtintab*, och trunkerar den till 16-bitar.

```
*code.insnp++ = (INSN)builtintab[GetSmall(arg1)];
```

builtin_rel(Integer) — tar fram en 32-bitsadress ur *builtintab* och beräknar relativa avståndet till denna.

```
*code.ulongp = (unsigned long)builtintab[GetSmall(arg1)]  -
code.ulong;
code.ulongp++;
```

builtin_offset(Integer, Offset) — tar fram en 32-bitsadress ur *builtintab* och beräknar relativa avståndet till denna. Adderar ett *Offset*.

```
*code.longp++ =
        (long)builtintab[GetSmall(arg1)]  + GetInteger(arg2);
```

relative(ClauseNo, Offset) — tar fram adressen till klausul *ClauseNo* ur nuvarande predikat. Adderar sedan ett offset.

```
*code.charpp =
        clause_adress(arg1) + GetSmall(arg2) - code.ulong;
code.charpp++;
```

8.4. SPARC-versionen

I SPARC-versionen är instruktionerna alltid 32-bitsord. Ofta kan denna bara kompileras till viss del. Därför lägger länkdata ofta till föregående 32-bitsord.

Makrot LO(Z) maskar av de 10 sista bitarna av ett ord och HI(Z) skiftar ned resterande 22 bitarna.

```
#define HI(Z) (Z >> 10)
#define LO(Z) (Z & 0x3FF)
```

short(Num) — tar fram 16-bitarstalet.

```
*code.shortp++ = (INSN)GetSmall(arg1);
```

add_lo_absolute(ClauseNo, Off) — tar fram den lägre delen av den absoluta adressen av en klausul i predikatet som kompileras. Lägger till ett offset. Detta adderas sedan till föregående 32-bitsord.

```
*(code.ulongp - 1) +=
        (unsigned long)LO(clause_adress(arg1) + GetSmall(arg2));
```

add_hi_absolute(ClauseNo, Off) — som ovan men tar fram den övre delen av adressen.

```
*(code.ulongp - 1) +=
        (unsigned long) HI(clause_adress(arg1) + GetSmall(arg2));
```

add_lo_builtin(Integer) — tar fram den lägre delen av värdet i tabellen, *builtintab*. Detta adderas sedan till föregående 32-bitsord.

```
*(code.ulongp - 1) +=
        LO((unsigned long)builtintab[GetSmall(arg1)]);
```

add_hi_builtin(Integer) — som ovan men tar fram den övre delen av värdet.

```
*(code.ulongp-1) += HI((unsigned long)builtintab[GetSmall(arg1)]);
```

add_lo_functor(Name, Arity) — tar fram den lägre delen av funktorn. Detta adderas sedan till föregående 32-bitsord.

```
*(code.ulongp-1) +=
        LO((unsigned long)SetArity(arg1,GetSmall(arg2)));
```

add_hi_functor(Name, Arity) — som ovan men tar fram den övre delen av funktorn.

```
*(code.ulongp-1) +=
        HI((unsigned long)SetArity(arg1,GetSmall(arg2)));
```

add_lo_tagged(Term) — tar fram den lägre delen av det taggade värdet. Detta adderas sedan till föregående 32-bitsord.

```
*(code.taggedp-1) += LO(arg1);
```

add_hi_tagged(Term) — som ovan men tar fram den övre delen av det taggade värdet.

```
*(code.taggedp-1) += HI(arg1);
```

call_builtin(Integer) — tar fram adressen till en subrutin och skapar en maskinkodsinstruktion som hoppar dit.

```
*code.ulongp =  ((unsigned long)((long)builtintab[GetSmall(arg1)]
             - code.long) >> 2) | NATIVE_CALL;
code.ulongp++;
```

call_native_entry(Predicate) — tar fram adressen till koden som ligger i ett predikats definition, ”struct definition” (se nedan, Snittet mot emulatorn – Anrop av predikat).

```
DerefHeap(arg1, arg1);
*code.ulongp = ((unsigned long)
             ((long)(&parse_definition(arg1, typein_mod)->entry)
             - code.long) >> 2) | NATIVE_CALL;
code.ulongp++;
```

call_absolute(ClauseNo, Off) — tar fram den absoluta adressen av en klausul i predikatet som kompileras. Base är klausulnummer. Lägger till ett offset. Skapar en instruktion som hoppar dit.

```
*code.ulongp = ((unsigned long)
          ((long)(clause_adress(arg1) + GetSmall(arg2))
          - code.long) >> 2) | NATIVE_CALL;
code.ulongp++;
```

9. Utökning av ql-formatet *n68_pred.p4 nsp_pred.p4*

Några instruktioner har tillkommit för att understödja maskinkodskompilering. Principen är som för andra ql-instruktioner – skapa ett objekt först och lämna ”hål” där inget kan fyllas i vid kompileringen. Efter dessa instruktioner kommer ql-instruktioner för att fylla hålen, oftast med symbolisk information.

Om samma värde ska in på flera ställen i kodobjektet och hålen är 32 bitar stora så läggs inte flera instruktioner ut. Istället länkas hålen ihop och instruktionen pekar bara ut den första i kedjan.

◊ Den maskinberoende delen av ql-generatorn finns i slutet på filen *nsp_asm.p4* och *n68_asm.p4.*

10. Snittet mot emulatorn *wam.c shdisp_r.c shdisp_w.c*

Snittet till emulatorn är utformat utifrån vissa kriterier:

- Få ändringar i nuvarande emulator
- Anrop från ett maskinkodspredikat till ett annat maskinkodspredikat ska vara mycket effektivt
- Emulatorn ska inte anropas rekursivt

Detta har uppnåtts genom ett meddelande-protokoll mellan emulatorn och maskinkods-utökningen. Maskinkod anropas genom att lämplig funktion anropas

execute_native anropas av emulatorn om ett predikat är i maskinkod. Viktigt att notera är att WAM-register CP, continuation, i emulatorfallet innehåller en pekare till byte-kod medan det i maskinkodsfallet är en hoppadress till exekverbar maskinkod.

För att maskinkodsutökningen inte ska hoppa till det CP som gäller vid inträdet så sparas det CP:et undan med en *allocate*. Ett nytt CP sätts att peka på en maskinkodsrutin som avallokerar den omgivningen och därmed återställer CP. Sedan lämnas kontrollen över till emulatorn.

retry_native anropas när emulatorn träffar på WAM-instruktionen *RETRY_NATIVE* vilken ligger först i varje maskinkodsklausul.

proceed_native anropas från emulatorn om den träffar på instruktionen *LEAVE* och vi vill göra en ”continuation” till maskinkod. Ett anrop till *deallocate* återställer korrekt CP som pekar på maskinkod.

WAM-register CP, continuation, i emulatorfallet är en pekare till byte-kod medan i maskinkodsfallet så är det en hoppadress till exekverbar maskinkod. För att emulatorn inte ska tolka det som CP pekar på som en WAM-instruktion så sparas det

CP:et undan med en *allocate*. Ett nytt CP sätts att peka på en liten WAM-rutin med en enda instruktion – *LEAVE*. Den avallokerar för att få maskinkods-CP igen.

◊ Detta protokoll kan resultera i att det finns redundanta par av omgivningar på lokala stacken, dvs par som tar ut varandra. Dessa par tas bort i *execute_native* och i *EXECUTE_EMUL* (se nedan).

När maskinkodsutökningen vill att emulatorn ska göra ett jobb så returnerar den funktion som anropade maskinkodsutökningen sist med en returkod

EXECUTE_EMUL signalerar till emulatorn att den ska köra ett emulerat predikat och sedan fortsätta i maskinkod. Emulatorn sparar undan maskinkods-CP och ersätter detta med ett CP som pekar på WAM-instruktionen *LEAVE*.

Det kan vara så att målet är i maskinkod men att emulatorn ”luras” att klara av alla events innan den upptäcker detta och anropar maskinkod igen.

FAIL_EMUL signalerar till emulatorn att en klausul var i byte-kod.

PROCEED_EMUL signalerar att en ”continuation” refererar till bytekod.

◊ En hel del data flyttas mellan minne och register vid inträde och utträde ur maskinkodsutökningen. Detta gör att vissa typer av program inte kommer att få någon uppsnabbning alls av att kompileras till maskinkod. Ett exempel är SICStus Prolog-interpretator.

10.1. Anrop av predikat

För att få maskinkodsanrop från en maskinkodsklausul till ett annat maskinkodspredikat så effektivt som möjligt har den struktur som är startpunkten till ett predikat utökats. I structuren ”struct definition” har maskinkod lagts. Idén är att undvika test på data och undvika att hämta data utan bara köra kod.

10.2. Anrop i SPARC-versionen

Ett anrop av ett maskinkodspredikat från en maskinkodsklausul i SPARC-versionen blir ett hopp till en funktor och vanligtvis ett vidarehopp till predikatkoden.

```
struct definition {
   short enter_instr;           /* see predtyp.h */
   short arity;
   struct mod_def *module;
   long entry;                  /* call predicate  or  call
event_emul */
   long dummy;      /* delay slot,  heap_warn_soft == heap_warn? */
   .
   .
};
```

Koden som finns i funktorn kan vara

- Ett normalt, snabbt, anrop till ett maskinkodspredikat.

```
call  predicate_indexing         ! normal case
cmp   ...                        ! test if (H > heap_warn_soft)
```

- Ett anrop till ett emulerat predikat. Hoppet sker till maskinkod som sköter om växlingen. Därför behövs inget test vilken kodtyp predikatet har. Det behövs inte heller sparas någon pekare till predikatet. Vid ett hopp sparas automatiskt adressen till hopp-instruktionen i ett register så detta kan användas för att återstarta anropet i emulatorn. Rutinen *execute_emul* sköter om växlingen mellan maskinkodsutökningen och emulatorn.

```
call  execute_emul              ! enter emulator
cmp   ...                       ! test if (H > heap_warn_soft)
```

- Ett anrop till ett predikat skrivet i C. Liknar förra fallet men färre saker sparas undan och anropet kan ske utan att emulatorn blandas in. Vi vet att ett predikat skrivet i C aldrig anropar emulatorn.

```
call  execute_c                 ! enter C predicate
cmp   ...                       ! test if (H > heap_warn_soft)
```

10.3. Anrop i 68K-versionen

68K-versionen liknar SPARC-versionen med skillnaden att inget hopp sker direkt från en klausul till kod i funktorn. I stället tas adressen som hoppet ska ske till fram ur fältet entry. Detta fält kan peka på koden för predikatet eller peka på *jsr*-instruktionen i funktorn.

```
struct definition {
   short enter_instr;           /* see predtyp.h */
   short arity;
   struct mod_def *module;
   short jsr;
   short exit;                  /* absolute short */
   INSN *entry;
   .
   .
};
```

Här blir fallen

- Ett normalt, snabbt, anrop till ett maskinkodspredikat. Detta använder ingen kod i funktorn.
- Ett anrop till ett emulerat predikat. Pekaren *entry* pekar då ut *jsr*-instruktionen så hoppet sker till denna. Instruktionen hoppar vidare till *execute_emul* sköter om växlingen mellan maskinkodsutökningen och emulatorn. Adressen till funktorn fås via returadressen som ligger på stacken.

```
jsr   execute_emul                  ! enter emulator
```

- Ett anrop till ett predikat skrivet i C.

```
jsr   execute_c                     ! enter C predicate
```

  - ◊ I NC68K-versionen så är hoppfälltet *exit* ett 16-bitars ord eftersom subrutin-biblioteket alltid kommer att länkas först och hamna på låga, kortare, adresser.

11. Subrutin-biblioteket *n68_pred.p4 nsp_pred.p4*

I subrutinbiblioteket ligger alla statiska rutiner. Det skulle ta aldeles för stor plats att beskriva dessa i detalj.

- Koden för alla WAM-instruktioner som skulle ta för stor plats att placeras i klausulerna. Till dessa hör bland andra allocate och get_value. Här finns också de inbyggda predikaten och inbyggda funktionerna.
- Rutiner som anropas vid fail.
- Rutiner som anropas vid try, dvs när en valpunkt skapas.
- Rutiner för *heap_overflow*.
- Gränssnittsrutiner.

12. Symboler

SICStus maskinkodsutökning är beroende av att information om konstanter, tabellindex (builtintab) stämmer överens mellan C-filer, assembler-filer och prolog-filer. För att underlätta detta så genereras konstanter automatiskt av programmen *nspmakem4.c*, *n68makem4.c* och *kernel_con.c*.

12.1. Assembler-konstanter

kernel_con.c kernel.con

En konstant *FOO* som används i både C och assembler ska vara definierad till samma värde på två ställen. I C nås fält i en *struct* via namn. I assembler används ett offset relativt starten av strukturen.

C-programmet *kernel_con* skapar en fil *kernel.con* som innehåller m4-makron som används i k*ernel_sp.mac*, *kernel_sp.s*, *kernel_68.mac* och *kernel_68.s*.

12.2. Gemensamma symboler och värden

nmakem4.c nc_builtin.c nc_builtin.h kernel.globl nsp_builtin.m4 n68_builtin.m4 nsp_builtin.pl n68_builtin.pl

Programmen *nspmakem4* och *n68makem4* läser filerna kernel_sp.s respektive kernel_68.s och producerar en hel rad filer som används av både av C, assembler och prolog.

Dels läses de subrutinadresser som startar med `__´och de speciella symboldefinitioner som finns i början av filen och startar med

`|G **var´**— index i *builtintab* för adressen till variabeln **var**.

```
m4:  define(__var, index)
C:      builtintab[index] = &var;
```

`|V **var = value´**— värdet av variabeln **var** är **value.**

```
m4:  define(__var, value)
```

`|**B** **var = value´**— symbolen **var** är index i *builtintab* där värdet **value** finns.

```
m4:  define(__var, index)
C:      builtintab[index] = value;
```

Filer som skapas är

nc_builtin.c — kod som fyller i element i *builtintab*. Detta kan vara konstanter såsom taggar och adressen till subrutiner i *kernel_*.s*. Dessa används sedan vid inlänkningen av prolog-program som kompilerats till maskinkod.

nc_builtin.h — *extern*-deklarationerna av subrutinerna ovan.

kernel.globl — *globl*-deklarationerna av de subrutiner i *kernel_*.s* som ska vara globala, dvs nås från C. Filen inkluderas av *kernel_*.s*.

nsp_builtin.m4 och **n68_builtin.m4** — makron som appliceras på n*_pred.p4, n*_clause.m4 och n*_asm.m4 för att skapa `.pl´-filer.

nsp_builtin.pl och **n68_builtin.pl** — prolog fakta som omvandlar ett index i builtintab till ett namn på rutinen eller tvärtom. Används inte men kan användas för att göra lästliga utskrifter av den symboliska assemblerkoden i prolog.

- ◊ Det finns en konstant i filen *kernel_sp.s* och *kernel_68.s* som måste ändras manuellt. Det är konstanten *trail_top* som ändras om fält läggs till i strukturen ”struct worker”.
- ◊ Predikaten *name_of_dunction/2* och *name_of_builtin/2* i filen *plwam.pl* omvandlar ett symboliskt namn till ett index i arrayen *builtintab* . I *builtintab* finns adressen till de C-funktioner som avses. Predikatet *nc_kernel/2* i filen *nsp_clause.p4* och *n68_clause.p4* används på motsvarande sätt av maskinkodsgeneratorn och måste manuellt uppdateras för att stämma med dessa tabeller.

13. Referencer

[Car91] *The Emulator for SICStus*, Mats Carlsson, SICS

[Car90:1] *Freeze, Indexing and Other Implementation Issues in the WAM*, Mats Carlsson, SICS R 86011B (revised 1990)

[Car90:2] *On the Efficiency of Optimising Shallow Backtracking in Compiled Prolog*, Mats Carlsson, SICS R 90003

[Bor91] *Improved Structure Copying in WAM,* Kent Boortz, SICS